UniCTest

Ingegneria del Software – A.A. 2021/2022

Progetto d’esame di: Blanco Francesco Giulio, Anzalone Dario Giuseppe

Fase di Elaborazione – Iterazione 4

Sommario

[1 Versione 2](#_Toc95092329)

[2 Introduzione e Elaborati 2](#_Toc95092330)

[3 Analisi Orientata agli Oggetti 2](#_Toc95092331)

[3.1 Modello di dominio 2](#_Toc95092332)

[3.1.1 Considerazione #1: Generalizzazione Template 4](#_Toc95092333)

[3.1.2 Considerazione #2: Applicazione del pattern GoF strutturale Decorator 4](#_Toc95092334)

[3.1.3 Modello di Dominio 4](#_Toc95092335)

[3.2 SSD 6](#_Toc95092336)

[3.3 Contratti delle Operazioni 6](#_Toc95092337)

[CO1 6](#_Toc95092338)

[CO2 7](#_Toc95092339)

[CO3 7](#_Toc95092340)

[CO4 7](#_Toc95092341)

[4 Progettazione Orientata agli Oggetti 7](#_Toc95092342)

[4.1 La classe Amministratore: Applicazione del pattern GoF Decorator 8](#_Toc95092343)

[4.1.1 Scenario 1 9](#_Toc95092344)

[4.1.2 Scenario 2 9](#_Toc95092345)

[4.2 Estensioni: Inserimento di una Materia nel Sistema e applicazione del pattern GoF Flyweight 10](#_Toc95092346)

[4.2.1 Struttura offerta dal pattern 11](#_Toc95092347)

[4.3 Diagrammi di interazione 13](#_Toc95092348)

[4.4 DCD 15](#_Toc95092349)

# Versione

|  |  |  |  |
| --- | --- | --- | --- |
| Versione | Data | Descrizione | Autori |
| Elaborazione 4 | 2022/01/29 | Quarta iterazione della fase di elaborazione. | Anzalone Dario Giuseppe, Blanco Francesco Giulio. |

# Introduzione e Elaborati

Nella 4° iterazione della fase di Elaborazione si è scelto di focalizzare l’attenzione su:

* Scenario di successo di UC2/A (introdotto nella seconda revisione della fase di Ideazione): Crea template di test ufficiale.
* Estensione del caso d’uso UC2/A.

Si prevede che questa sarà una *breve iterazione*.

Si creeranno/aggiorneranno i seguenti artefatti:

* Nel contesto dell’Analisi Orientata agli Oggetti:
  + **Modello di Dominio** (da aggiornare)
  + **SSD (System Sequence Diagrams)**
  + **Contratti delle operazioni**
* Nel contesto della Progettazione Orientata agli Oggetti:
  + **Diagrammi di interazione**
  + **DCD (Design Class Diagram)** (da aggiornare)

La 4° iterazione della fase di elaborazione prevederà tre brevi fasi di:

1. Analisi orientata agli oggetti (OOA).
2. Progettazione orientata agli oggetti (OOD).
3. Implementazione.

# Analisi Orientata agli Oggetti

## Modello di dominio

Si riporta lo scenario di successo di UC2/A. Vengono evidenziate le locuzioni nominali per la scelta delle classi concettuali candidate. Le classi concettuali candidate verranno evidenziate in verde, gli attributi in giallo.

**UC2/A. Crea template di test ufficiale**

**Nota del cliente:** l’Amministratore è un Tutor con diritti maggiori rispetto ad un Tutor classico. L’Amministratore, oltre a insegnare materie (dunque inserire nuovi quesiti), può anche inserire nuovi Tutor nel Sistema, oltre che inserire Studenti e template ufficiali. I diritti di Amministratore potranno essere assegnati o rimossi a qualunque Tutor in qualunque momento.

Nel caso d’uso e in generale ci si riferirà al Tutor con i diritti di Amministratore semplicemente come Amministratore.

|  |  |
| --- | --- |
| Scenario principale di successo | 1. L’Amministratore vuole creare un nuovo template.  2. L’Amministratore sceglie l’attività “Crea template ufficiale” e inserisce il nome del template ufficiale che deve inserire nel Sistema. Il Sistema registra le informazioni inserite.  3. L’Amministratore inserisce:   * la fonte che ha stabilito le regole del test su cui il template ufficiale deve basarsi * il formato dei quesiti (cioè il numero di risposte e il numero di risposte corrette per quesito) * il punteggio attribuito ad un singolo quesito (nel caso di risposta corretta, risposta errata e risposta non data) * il tempo totale previsto dal test ufficiale.   Il Sistema registra le informazioni inserite.  4. L’Amministratore inserisce il nome della materia che deve essere inserita nel Test. Il Sistema crea una nuova sezione del template relativa alla materia inserita.  5. L’Amministratore seleziona il numero di quesiti previsti per la materia selezionata. Il Sistema registra le informazioni inserite.  *I passi 4, 5 e 6 vengono ripetuti finché serve.*  6. L’Amministratore indica di aver finito. |
| Estensioni | **5a.** L’Amministratore ha inserito nel Sistema una materia non ancora presente in esso. La nuova materia viene aggiunta nel Sistema. |

Le classi concettuali candidate sono:

* **Amministratore**: è un tutor con più diritti rispetto ad un tutor classico (possiede diritti di amministrazione, tra cui inserimento di nuovi template ufficiali).
* **TemplateUfficiale**: contiene informazioni di fonti ufficiali relative ad un test di ammissione.

### Considerazione #1: Generalizzazione Template

Si può notare che TemplateUfficiale presenta gli stessi attributi di TemplatePersonalizzato (classe concettuale introdotta nell’iterazione 2 della fase di Elaborazione, analizzando UC2) ad eccezione di:

* **fonte**, attributo inesistente in TemplatePersonalizzato, essendo il TemplatePersonalizzato creato dallo stesso Studente
* **tempoTotale**, indica il tempo totale di svolgimento del test. Viene introdotto al posto del tempoMedio del TemplatePersonalizzato, che indicava il tempo medio di svolgimento di un solo quesito.

Risulta dunque chiaro che non sia necessario introdurre una nuova classe TemplateUfficiale ex-novo, in quanto condivide la maggior parte delle informazioni del TemplatePersonalizzato. Dunque, si è deciso di rappresentare i due template come **specializzazione** di una **classe concettuale astratta** ***Template***.

La classe astratta *Template* avrà tutti gli attributi ad eccezione degli attributi non condivisi tra il TemplatePersonalizzato e il TemplateUfficiale (ovvero: fonte, tempoTotale, tempoMedio), che saranno specifici delle classi specializzazione.

### Considerazione #2: Applicazione del pattern GoF strutturale Decorator

Alla luce della nota contenuta in UC2/A, bisogna concludere che l’Amministratore non sia una classe candidata nuova, bensì possa essere vista come una responsabilità addizionale di un tutor classico. Ciò è testimoniato dal fatto che questa responsabilità può essere assegnata o rimossa a *runtime*.

Non si può non considerare che una nota simile sia presente in riferimento al Tutor di simulazione nella seconda revisione della fase di Ideazione.

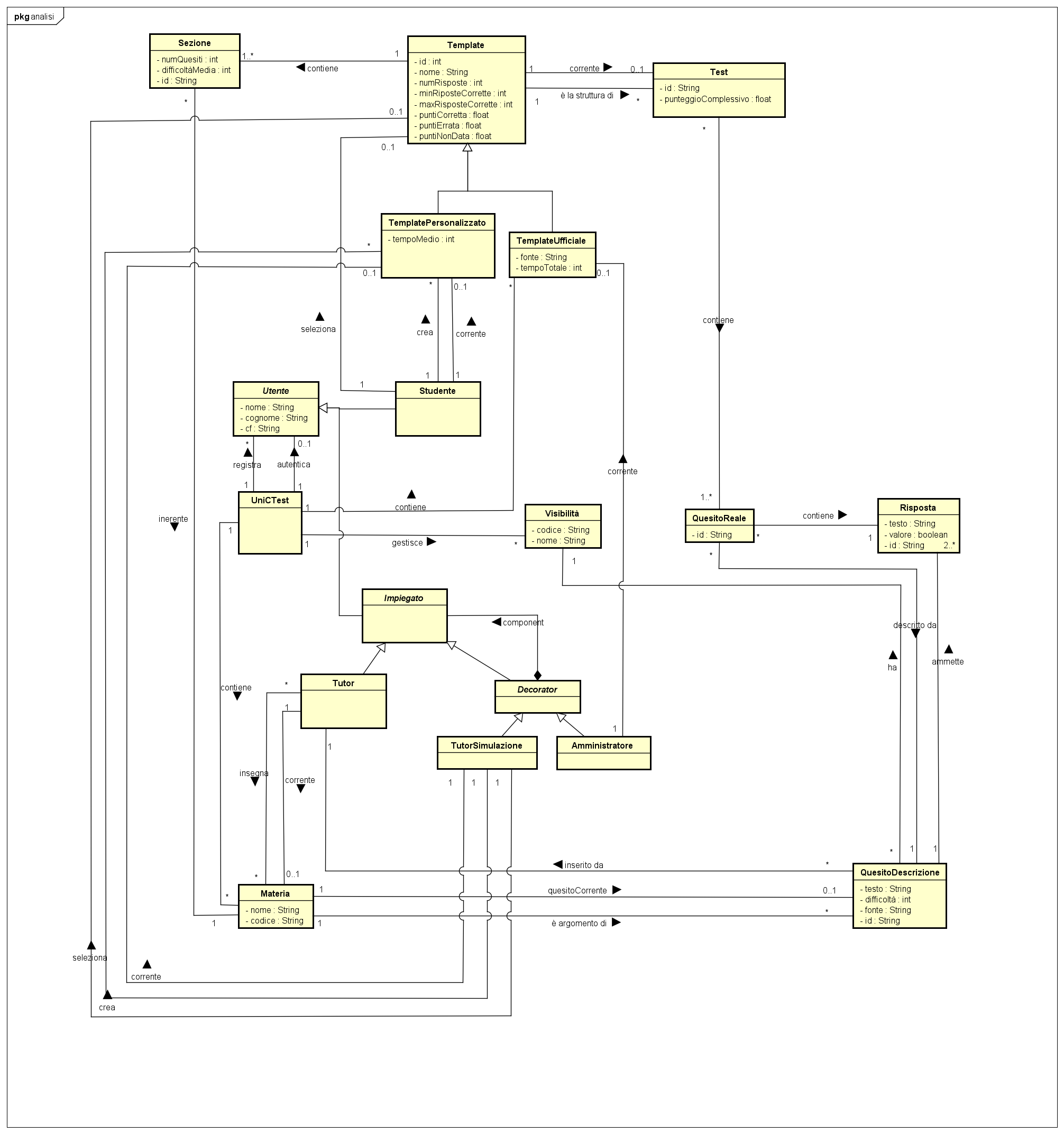
Quindi, si può concludere che il Tutor può essere *decorato* con responsabilità addizionali (ad esempio responsabilità di Amministratore – che verrà introdotta in questa sede – oppure altre responsabilità, come quella di Tutor di simulazione – che verranno introdotte in iterazioni successive).

Ciò suggerisce l’applicazione del **pattern GoF Decorator**.

Nella fase di Progettazione verrà esplorata meglio la soluzione offerta dal pattern.

### Modello di Dominio

Dalle classi concettuali che sono state ricavate e dalle considerazioni appena effettuate è stato ricavato il seguente Modello di Dominio, comprendente anche le classi concettuali ricavate nelle fasi di OOA relative alle precedenti iterazioni:



## SSD

## Contratti delle Operazioni

Di seguito si riportano le descrizioni delle Operazioni eseguite in riferimento a UC2/A.

Nota: per distinguere le operazioni di gestione del TemplateUfficiale da quelle di gestione del TemplatePersonalizzato da parte dello Studente, sono state prese le seguenti misure:

* le operazioni effettuate dall’attore Amministratore per inserire un TemplateUfficiale rispetteranno la notazione *<nomeOperazione>U*.
* le operazioni effettuate dall’attore Studente per creare un TemplatePersonalizzato (in relazione a UC2, iterazione 2 della fase di Elaborazione) sono state rinominate in *<nomeOperazione>P*.

### CO1

|  |  |
| --- | --- |
| Operazione | nuovoTemplateU(nome: String) |
| Riferimenti | UC2/A: Crea template di test ufficiale |
| Pre-condizioni | - |
| Post-condizioni | * È stata creata l’istanza *tu* di TemplateUfficiale con *tu.nome*=nome ed è stata associata all’Amministratore tramite l’associazione “corrente”. |

### CO2

|  |  |
| --- | --- |
| Operazione | inserisciInfoTemplateU(fonte: String, puntiCorretta: float, puntiErrata: float, puntiNonData: float, tempoTotale: int, numRisposte: int, minRisposteCorrette: int, maxRisposteCorrette: int) |
| Riferimenti | UC2/A: Crea template di test ufficiale |
| Pre-condizioni | È in corso la creazione di un TemplateUfficiale *tu*. |
| Post-condizioni | * Sono stati inizializzati gli attributi di *tu*. |

### CO3

|  |  |
| --- | --- |
| Operazione | creaSezioneU(nomeMateria: String, numQuesiti int) |
| Riferimenti | UC2/A: Crea template di test ufficiale |
| Pre-condizioni | È in corso la creazione di un TemplateUfficiale *tu*. |
| Post-condizioni | * È stata inizializzata l’istanza *s* di Sezione ed è stata associata a *tu* tramite l’associazione “contiene”. * L’istanza *s* è stata associata all’istanza *m* di Materia avente l’attributo *m.codice* uguale a codiceMateria. |

### CO4

|  |  |
| --- | --- |
| Operazione | confermaTemplateU() |
| Riferimenti | UC2/A: Crea template di test ufficiale |
| Pre-condizioni | È in corso la creazione di un TemplateUfficiale *tu*. |
| Post-condizioni | * L’istanza *tu* è stata associata all’istanza *a* dell’Amministratore tramite l’associazione “crea”. |

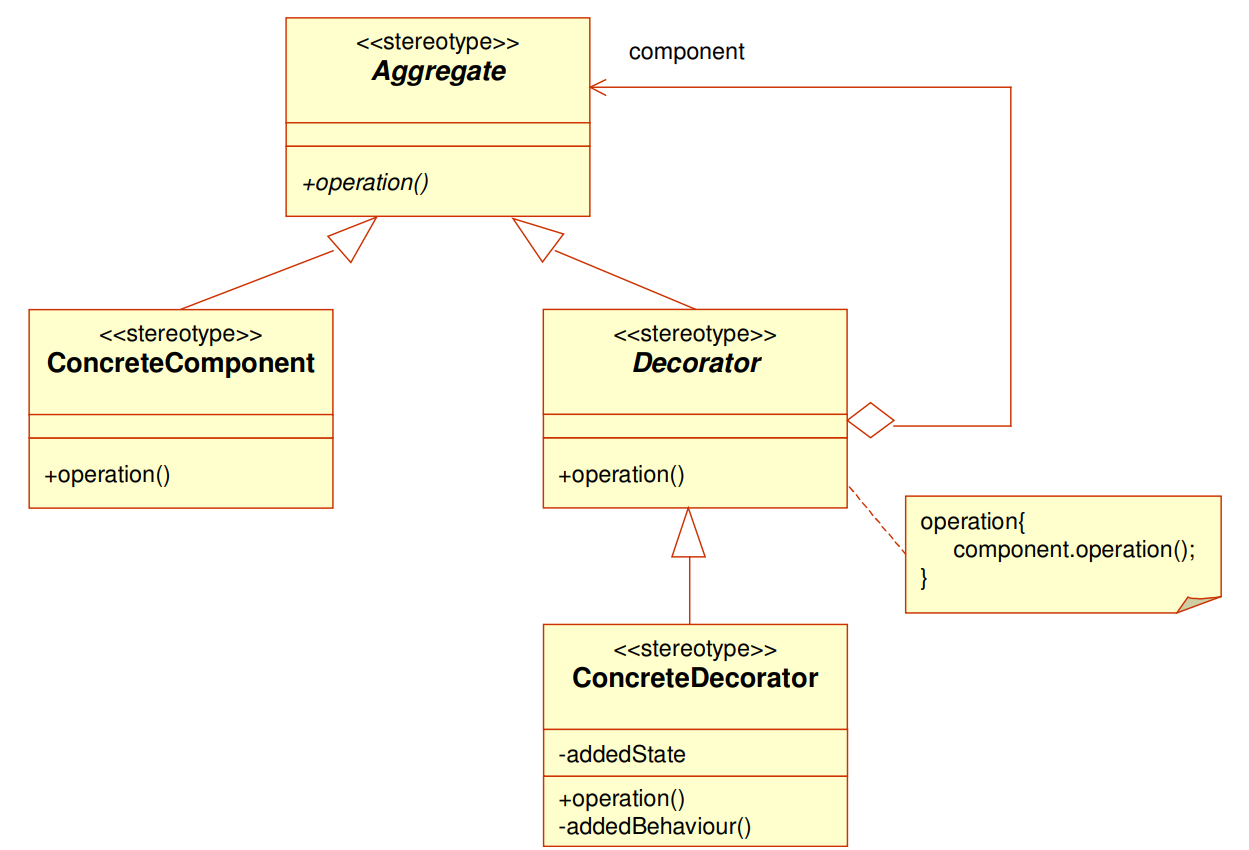
# Progettazione Orientata agli Oggetti

Aggiornate le classi concettuali nel contesto della OOA, si passa alla OOD, realizzando il Modello di Progetto ed in particolare aggiornando il DCD (visualizzazione statica delle classi software), aggiornamento da effettuare il parallelo alla realizzazione dei diagrammi di interazione (visualizzazione dinamica delle classi software).

L’Analisi Orientata agli Oggetti condotta in questa iterazione ha portato all’inserimento del pattern Decorator, che qui viene meglio illustrato.

## La classe Amministratore: Applicazione del pattern GoF Decorator

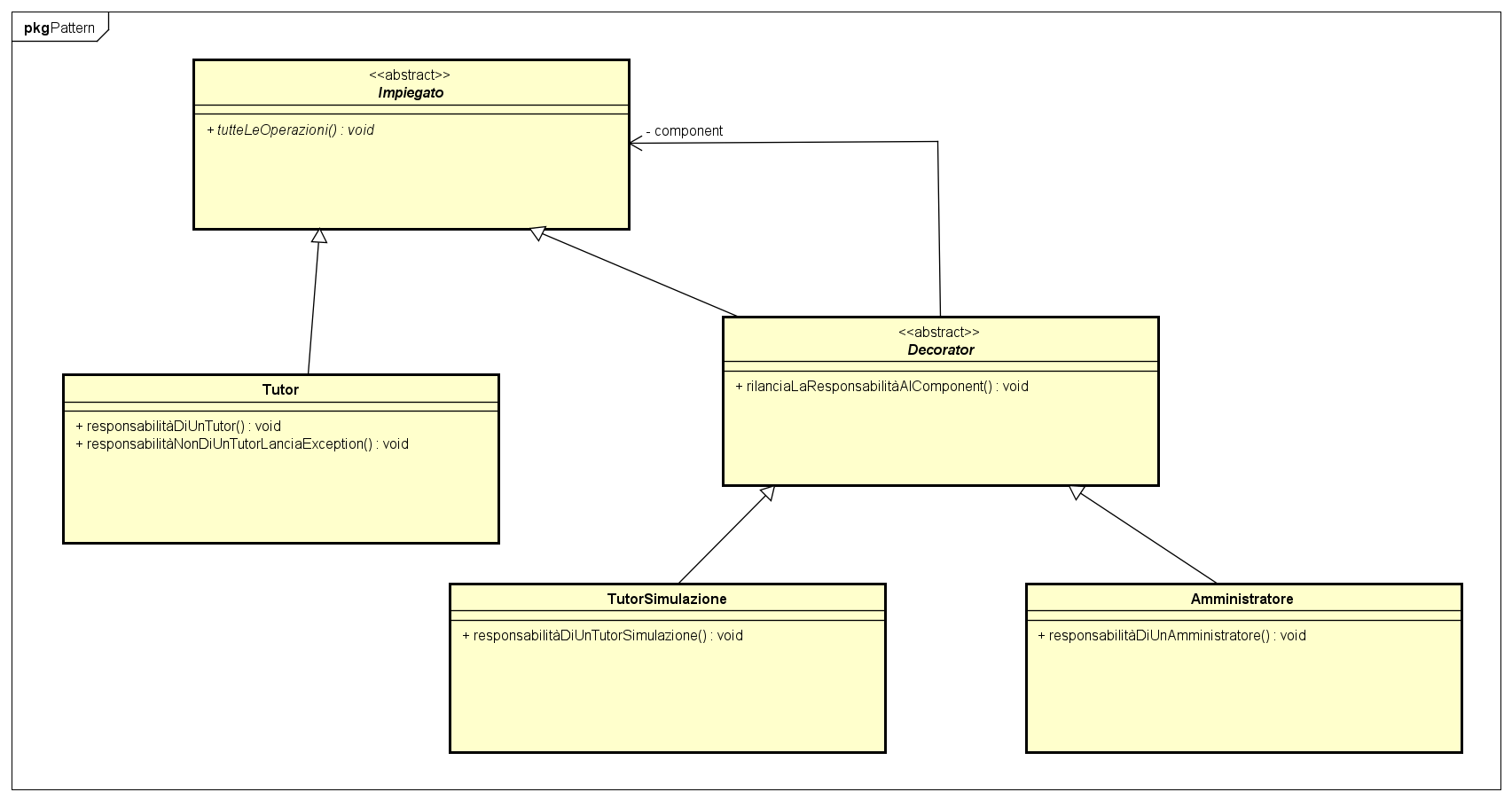
Struttura offerta dal pattern:



La struttura (nel nostro caso) prevede l’impiego di:

* Classe astratta **Aggregate**: risponde all’esigenza di avere un’interfaccia comune per gli oggetti eventualmente decorati di responsabilità aggiuntive.
* Classe **ConcreteComponent**: implementa l’oggetto a cui possono essere aggiunte o rimosse delle responsabilità.
* Classe astratta **Decorator**: risponde all’esigenza di avere un’interfaccia comune per i decoratori. Un decorator ha il riferimento ad un Aggregate, che a sua volta può essere un Decorator oppure un ConcreteComponent.
* Classe **ConcreteDecorator**: è la classe che concretamente aggiunge le responsabilità al ConcreteComponent.

Nel contesto di UniCTest:

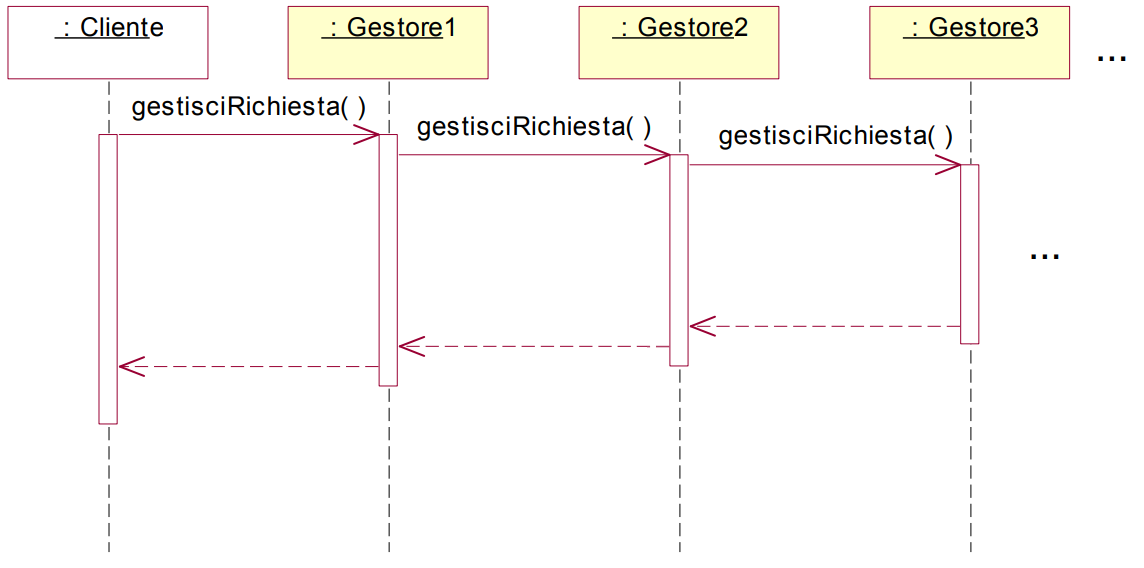
* La classe **Aggregate** è la classe **Impiegato**.
* La classe **ConcreteComponent** è la classe Tutor. Infatti, il Tutor è la classe da arricchire di responsabilità addizionali.
* La classe **Decorator** mantiene invariato il nome. Decorator generalizza Amministratore e si prevede che nelle prossime iterazioni generalizzerà anche TutorSimulazione.
* La classe **ConcreteDecorator** è la classe **Amministratore** (e si prevede che un altro ConcreteDecorator sarà la classe TutorSimulazione).

La struttura del pattern, così come si evince dalla figura, permette di avere un Tutor che gode di più responsabilità addizionali assegnate e/o rimosse a *runtime*.

È stato elaborato un meccanismo che permette ad un Impiegato decorato di soddisfare qualunque responsabilità gli è concessa per via di qualunque sua decorazione.

Il meccanismo elaborato sfrutta il pattern GoF comportamentale **Chain of Responsibility**.

Il pattern Chain of Responsibility consente di applicare questo principio: se un gestore riesce a soddisfare una richiesta, allora la esegue, altrimenti passa la richiesta al gestore successivo.



In questo caso con “gestore successivo” si intende un gestore “più interno”, cioè il *component* di un Decorator.

Di seguito viene spiegato tale meccanismo considerando le due tipologie di scenari che possono verificarsi.

### Scenario 1

Si supponga l’esistenza di un’istanza *t1* = TutorSimulazione(Amministratore(Tutor))), ovvero un Tutor che è stato dapprima decorato con responsabilità di Amministratore, poi anche con responsabilità di TutorSimulazione. Qualora *t1* decida di eseguire una responsabilitàDiUnAmministratore(), allora verrebbe chiamata su un’istanza di TutorSimulazione che, non avendo implementato il metodo, chiamerebbe il metodo del proprio Decoratore. Tale metodo del Decoratore rilancerebbe la responsabilità al component più interno, cioè ad Amministratore(Tutor) che, essendo una istanza di Amministratore, è in grado di soddisfare la responsabilitàDiUnAmministratore() senza rilanciarla (infatti implementa tale metodo senza richiamarlo su un component più interno).

Quindi in tal caso la responsabilità è stata correttamente assegnata e soddisfatta.

### Scenario 2

Si supponga l’esistenza di un’istanza *t2* = TutorSimulazione(Tutor)), ovvero un Tutor che è stato decorato con responsabilità di TutorSimulazione ma non di Amministratore. Qualora *t2* decida di eseguire una responsabilitàDiUnAmministratore(), allora verrebbe chiamata su un’istanza di TutorSimulazione che, non avendo implementato il metodo, chiamerebbe il metodo del proprio Decoratore. Tale metodo del Decoratore rilancerebbe la responsabilità al component più interno, cioè a Tutor che, essendo un Tutor semplice e non decorato, non può soddisfare alcuna responsabilità addizionale. Quindi il Tutor può lanciare un’eccezione (dichiarando, dunque, di non aver soddisfatto la responsabilità perché *t2* non ne aveva il diritto).

Quindi in tal caso la responsabilità è stata correttamente non soddisfatta.

## Estensioni: Inserimento di una Materia nel Sistema e applicazione del pattern GoF Flyweight

Dal caso d’uso si evince che il Sistema non deve restituire all’Amministratore le materie disponibili, pertanto è possibile che l’Amministratore digiti una materia non ancora presente nel Sistema (magari perché non è ancora stato inserito alcun Tutor che la insegna).

L’inserimento di un template ufficiale può dunque causare l’inserimento di una nuova materia nel sistema.

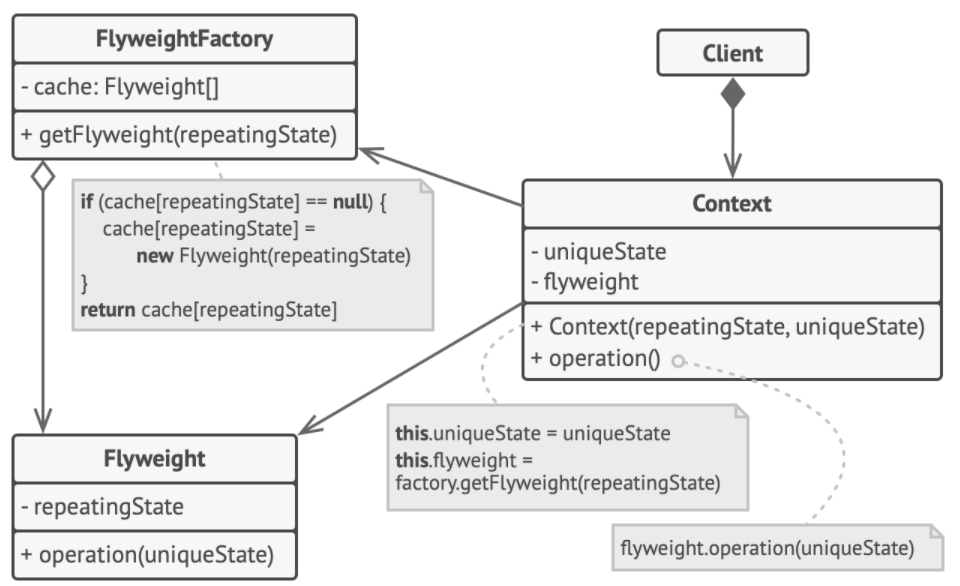
Tuttavia, se una materia è già presente nel Sistema, allora l’istanza di Materia da adoperare è quella già presente nel Sistema. Infatti, non avrebbe senso creare una nuova istanza di Materia, anche perché le materie sono deputate al contenimento dei quesiti e non si vuole che quesiti di uno stessa Materia risultino “sparsi” su varie liste di varie istanze della classe Materia.

Il pattern Flyweight è un pattern GoF strutturale mirato ad ottimizzare la memoria.

Di seguito viene analizzata la soluzione proposta dal pattern e viene analizzato perché tale soluzione possa risultare utile in questo contesto.

### Struttura offerta dal pattern

Il pattern prevede la seguente struttura:

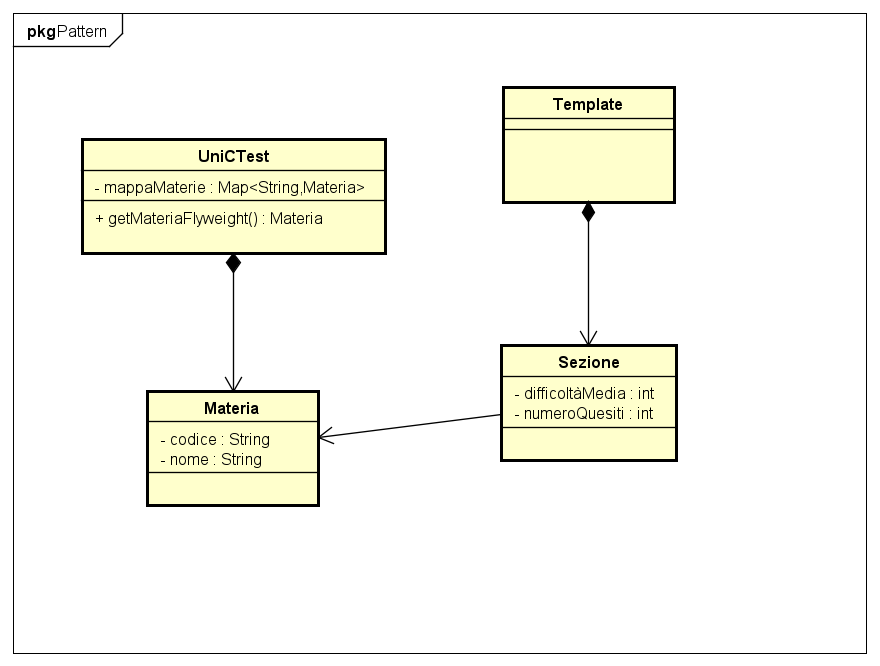


La struttura prevede l’impiego di:

* Classe **Flyweight**: contiene la porzione dell’oggetto originale che può essere condivisa tra più oggetti. Lo stesso oggetto flyweight può essere usato in diversi contesti. Il Flyweight contiene una parte dello stato dell’oggetto originale, detta parte ***intrinseca***, cioè quella parte che può essere condivisa perché è effettivamente comune a più oggetti. Lo stato dell’oggetto originale, non condiviso da vari oggetti, viene esternato dal flyweight e per questo è detto stato***estrinseco***. Tale stato viene aggiunto nella classe Context.
* Classe **Context**: contiene lo stato estrinseco dell’oggetto originale. Se la classe Context fosse un tutt’uno con la classe Flyweight, il pattern godrebbe di scarsa applicabilità in quanto non consentirebbe un notevole risparmio di memoria.
* Classe **Client**: in genere calcola lo stato estrinseco del flyweight. Dalla prospettiva del Client, il flyweight è un oggetto che può essere configurato a *runtime* a partire da qualche parametro di contesto.
* Classe **FlyweightFactory**: si occupa di creare, conservare, gestire e restituire gli oggetti flyweight.

Nel contesto di UniCTest:

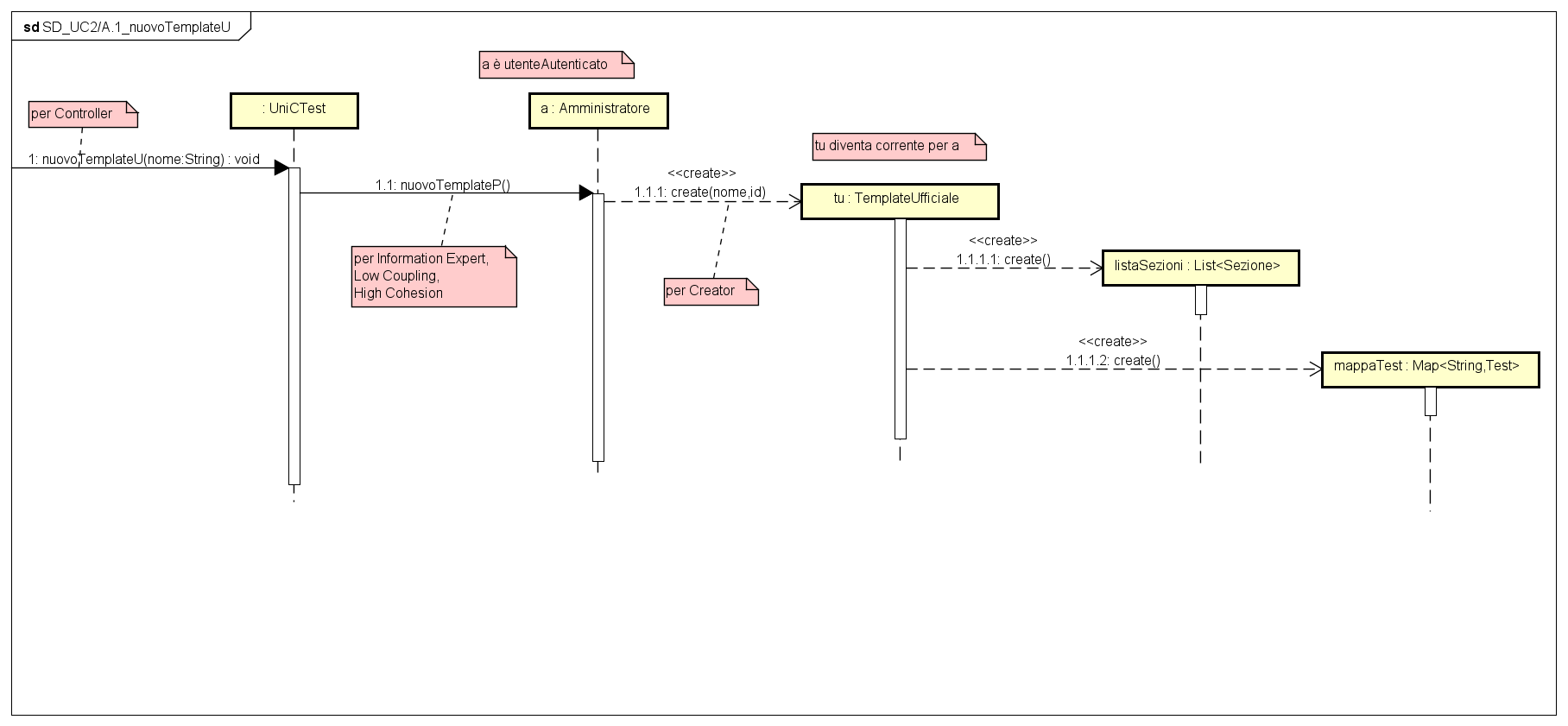
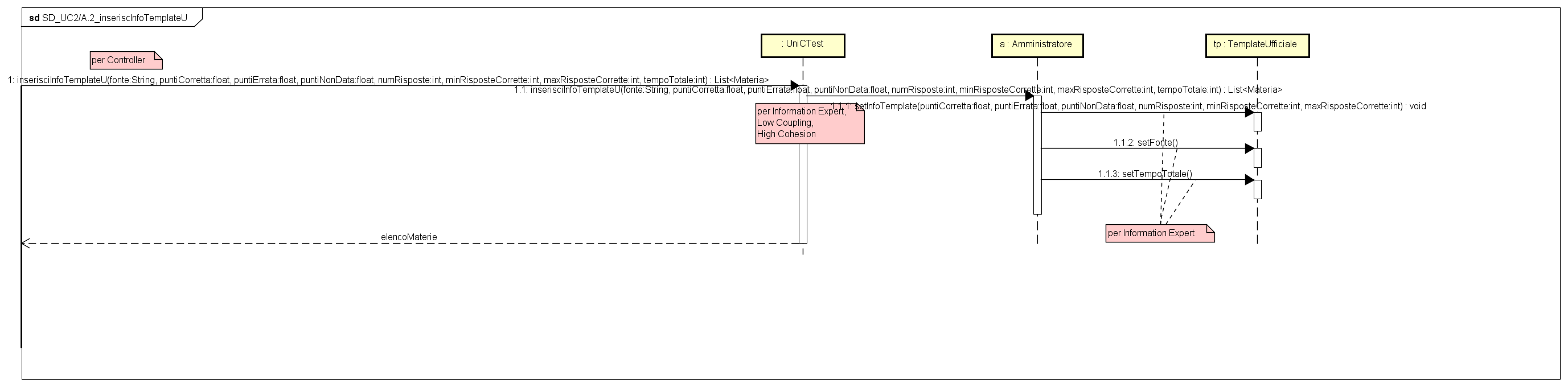
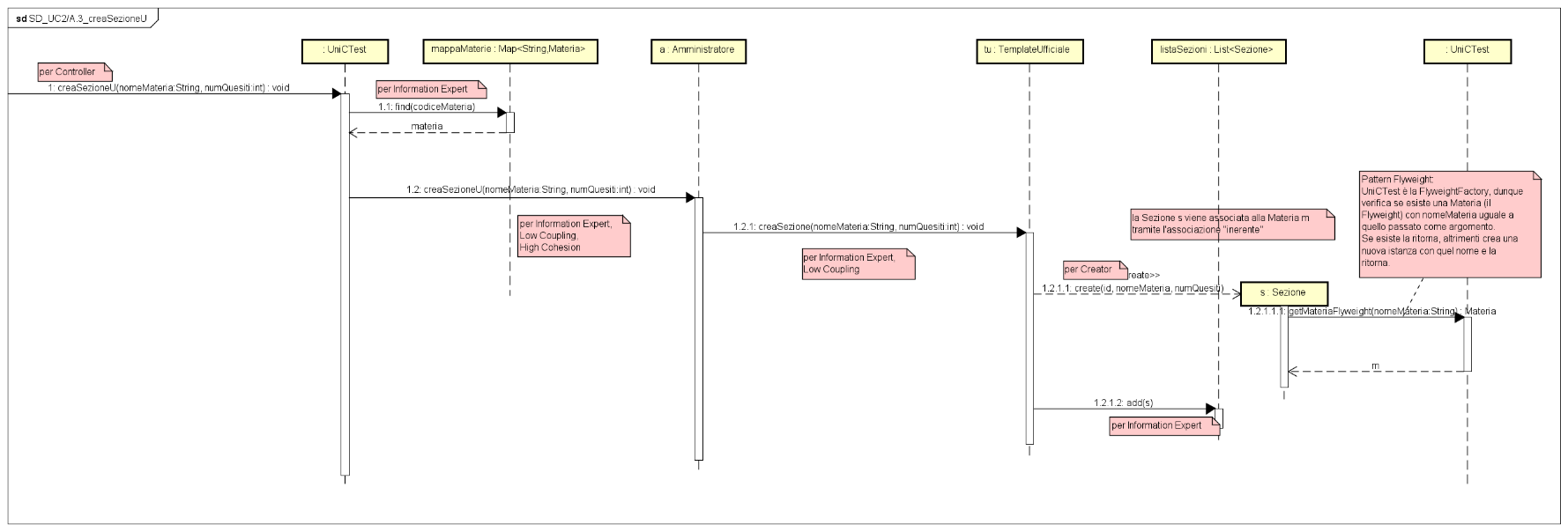
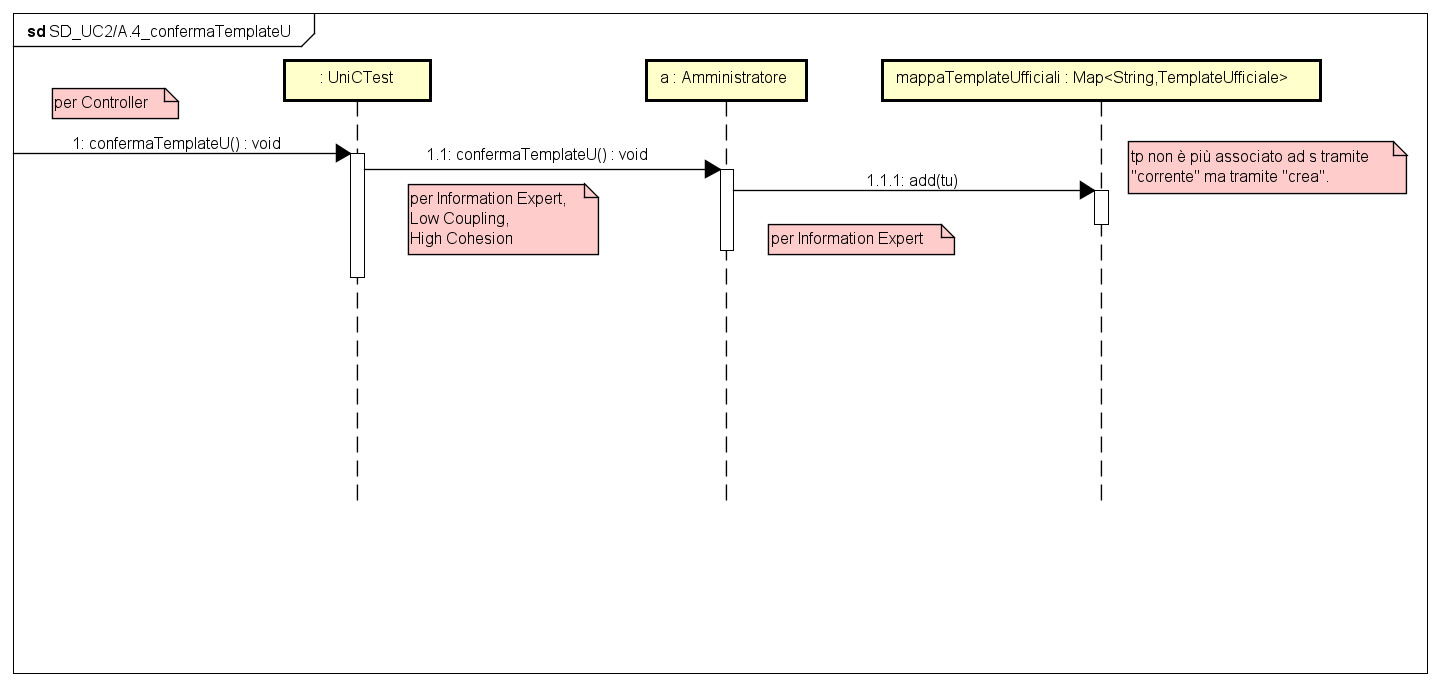
* La classe **Flyweight** è la classe **Materia**.
* La classe **Context** è la classe **Sezione**. Infatti, le Sezioni possiedono argomenti con una variabilità troppo ampia per poter essere considerate parte dello stato *intrinseco*.
* La classe **Client** è la classe **Template**. Il Template si occupa di effettuare lo storage e la gestione degli stati *estrinseci* della Materia, cioè delle Sezioni.
* La classe **FlyweightFactory** è la classe **UniCTest**. UniCTest crea, memorizza, gestisce e restituisce gli oggetti flyweight in maniera opportuna, cioè creandoli solo se essi non sono presenti nel Sistema.



Da notare che la creazione dell’oggetto flyweight si origina dalla richiesta di creazione di una nuova Sezione (Context) da parte del Template (Client), in particolare all’interno del costruttore della Sezione (Context).

Inoltre, l’associazione tra Context e FlyweightFactory, in questo specifico caso, non è necessaria, in quanto la FlyweightFactory è un **Singleton** (infatti UniCTest rappresenta l’intero Sistema software), dunque la sua istanza può essere recuperata mediante la chiamata al metodo statico *getInstance()*.

## Diagrammi di interazione

1. **SD\_UC2/A\_creaTemplateU**
2. **SD\_UC2/A\_inserisciInfoTemplateU**
3. **SD\_UC2/A\_creaSezioneU**
4. **SD\_UC2/A\_confermaTemplateU**

## DCD

